به نام خدا


تمرین 3 طراحی زبان‌های برنامه‌سازی

جناب آقای دکتر ایزدی

سارا آذرنوش

98170668

(1

Abstract syntax:

مجموعه ای از درختان که برای نشان دادن برنامه ها استفاده می شود. این بدین ترتیب است که نحوه برنامه را از نظر کامپایلر تعریف می کند.

(مانند درخت است.)

Concrete syntax:

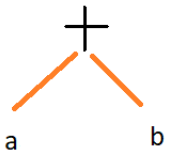
توسط گرامر تعریف می شود. شامل مجموعه ای از قوانین (تولیدات) که نحوه برنامه ها را برای برنامه نویس تعریف می کند.

(مانند string است.)

اگر جمع دو عدد ساده را درنظر بگیریم:

Abstract syntax:

Exp -> Exp -> Exp




Concrete syntax:

infix : a + b

prefix: (+ a b)

postfix: (a b +)

stack: push 2

push 3

add

accumlator: load a

add b

...

```
#lang racket
(define-struct queue (head tail len))
(define  empty-queue (queue '() '() 0))
(define (isempty? q)
  (null? (queue-head q)))

;n = queue size
(define (isfull? q n)
  (equal? (queue-tail q) (+ 1 n)))

(define (enqueue q x)
  (if (null? (queue-head q))
      (queue (reverse (cons x (queue-tail q))) '() (+ (queue-len q) 1))
      (queue (queue-head q) (cons x (queue-tail q))(+ (queue-len q) 1))))



(define (dequeue q)
  (cond [(empty? q) (error "empty")]
        [(not (null? (queue-head q)))
         (if (null? (rest (queue-head q)))
             (queue (reverse (queue-tail q)) '() (- (queue-len q) 1))
             (queue (rest (queue-head q)) (queue-tail q) (- (queue-len q) 1)))]
        [else (queue (reverse (queue-tail q)) '() (- (queue-len q) 1))]))

(define (peek q)
  (cond [(empty? q) (error "empty")]
        [(car (queue-head q))]))
```

```
#lang racket
(define-struct list (head tail len) #:mutable #:transparent)
(define-struct link (value previous next) #:mutable #:transparent)

(define (empty-doubly-linked-list) (list '() '() 0))

(define (isempty? list)
  (null? (list-head list)))

;n = listsize
(define (isfull? list n)
  (equal? (list-len list) (+ 1 n)))

(define (insert list before after value)
  (define new-link (make-link value before after))
  (if before
      (set-link-next! before new-link)
      (set-list-head! list new-link))
  (if after
      (set-link-previous! after new-link)
      (set-list-tail! list new-link))
  (set-list-len! list (+ (list-len list) 1))
    new-link)

(define (insert-before list link value)
  (insert list (link-previous link) link value))
```

```
(define (insert-after list link value)
  (insert list link (link-next link) value))

(define ( insert-first list value)
  (insert list #f (list-head list) value))

(define (delete list link)
  (let ((before (link-previous link))
        (after (link-next link)))
    (if before
        (set-link-next! before after)
        (set-list-head! list after))
    (if after
        (set-link-previous! after before)
        (set-list-tail! list before)))
  (set-list-len! list (- (list-len list) 1))
  )

(define (delete-first list)
  (delete list (list-head list)))
```

```
#lang racket
(define empty-env
  (lambda () '()))

(define empty?
  (lambda (e)
    (null? e)
    empty-error))

(define has-binding?
  (lambda (envi search-var)
    (letrec ((loop (lambda (env)
                (cond ((null? env)
                       #f)
                      ((and (pair? env) (pair? (car env)))
                       (let ((saved-var (caar env))
                             (saved-env (cdr env)))
                         (or (eqv? search-var saved-var) (loop saved-env))))
                      (else
                       (report-invalid-env envi))))))
      (loop envi))))

;env -> variables values
(define union
  (lambda (vars vals env)
    (cond ((and (null? vars) (null? vals))
           env)
          ((and (pair? vars) (pair? vals))
```

```
        (union (cdr vars) (cdr vals) (extend-env (car vars) (car vals) env)))
       ((null? vars)
        (no-var))
       (else
        (no-val)))))

;book
(define extend-env
  (lambda (var val e)
    (cons (cons var val) e)))



(define apply-env
  (lambda (env search-var)
              (cond
                ((eqv? (car env) 'empty-env)
                    (report-no-binding-found search-var env))
                   ((eqv? (car env) 'extend-env)
                    (let ((saved-var (cadr env))
                          (saved-val (caddr env))
                          (saved-env (cadddr env)))
                      (if (eqv? search-var saved-var)
                          saved-val
                          (apply-env saved-env search-var))))
                   (else
                    (report-invalid-env env)))))



(define (empty-error)
```

```
    (error 'not-set-variable))

(define no-var
  (lambda ()
    (error "no var")))

(define no-val
  (lambda ()
    (error "no value")))

;book
(define report-no-binding-found
  (lambda (search-var env)
    (error 'apply-env "No binding for ~s in ~s" search-var env)))

(define report-invalid-env
  (lambda (env)
    (error 'apply-env "Bad environment ~s" env)))
```

(5

```
#lang eopl
(define-datatype env env?
  (empty-env)
  (extend-env
   (saved-var symbol?)
   (saved-val (lambda (x) #t))
   (saved-env env?)))

(define has-binding?
  (lambda (e search-var)
```

```
    (cases env e
        (empty-env () #f)
        (extend-env (saved-var saved-val saved-env)
                (if (eqv? search-var saved-var)
                  #t
                  (has-binding? saved-env search-var))))))

(define apply-env
  (lambda (e search-var)
    (cases env e
        (empty-env ()
                (report-no-binding-found search-var))
        (extend-env (saved-var saved-val saved-env)
                (if (eqv? search-var saved-var)
                  saved-val
                  (apply-env saved-env search-var))))))


(define report-no-binding-found
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))
```